Deductive Software Verification—The KeY Book

From Theory to Practice

Springer
Chapter 5
Theories

Peter H. Schmitt and Richard Bubel

5.1 Introduction

For a program verification tool to be really useful it needs to be able to reason about at least the most important data types, both abstract data types and those built into the programming language. In Section 5.2 below the theory of finite sequences of arbitrary objects, allowing in particular arbitrary nestings of sequences in sequences, is presented. This presentation covers axioms for a core theory plus definitional extensions plus consistency considerations.

Section 5.3 contains an axiomatization of Java’s String data type. The theory of the mathematical integers has already been dealt with in Subsection 2.4.2. Section 5.4 below explains how the KeY system deals with the Java integer data types.

5.2 Finite Sequences

This section develops and explains the theory $T_{\text{seq}}$ of finite sequences. By $\text{Seq}$ we denote the type of finite sequences. The vocabulary $\Sigma_{\text{seq}}$ of the theory is listed in Figure 5.1. We will start with a simple core theory $\text{CoT}_{\text{seq}}$ and incrementally enrich it via definitional extensions. Typing of a function symbol $f$ is given as $f : A_1 \times \ldots \times A_n \rightarrow R$ with argument types $A_i$ and result type $R$, typing of a predicate symbol $p$ as $p(A_1 \times \ldots \times A_n)$.

Our notion of a sequence is rather liberal, e.g., $\langle 5, 6, 7, 8 \rangle$ is a sequence, in fact a sequence of integers. But the heterogeneous and nested list $\langle 0, \langle 0, \text{seqEmpty}, \text{null} \rangle, \text{true} \rangle$ is also allowed.

The semantics of the symbols of the core theory will be given in Definition 5.3. We provide here a first informal account of the intended meaning. The function value $\text{seqLen}(s)$ is the length of list $s$. Since for heterogeneous lists there is no way the type of an entry can be recovered from the type of the list, we provide a family of access functions $\text{seqGet}_A$ that yields the cast to type $A$ of the $i$-th entry in list $s$. 

Core Theory

\[ A::\text{seqGet} : \text{Seq} \times \text{int} \rightarrow A \quad \text{for any type } A \subseteq \text{Any} \]
\[ \text{seqGetOutside} : \text{Any} \]
\[ \text{seqLen} : \text{Seq} \rightarrow \text{int} \]

Variable Binder

\[ \text{seqDef} : \text{int} \times \text{int} \times \text{Seq} \rightarrow \text{Seq} \]

Definitional Extension

\[ \text{seqDepth} : \text{Seq} \rightarrow \text{int} \]
\[ \text{seqEmpty} : \text{Seq} \]
\[ \text{seqSingleton} : \text{Any} \rightarrow \text{Seq} \]
\[ \text{seqConcat} : \text{Seq} \times \text{Seq} \rightarrow \text{Seq} \]
\[ \text{seqSub} : \text{Seq} \times \text{int} \times \text{int} \rightarrow \text{Seq} \]
\[ \text{seqReverse} : \text{Seq} \rightarrow \text{Seq} \]
\[ \text{seqIndexOf} : \text{Seq} \times \text{Any} \rightarrow \text{int} \]
\[ \text{seqNPerm} : \text{Seq} \]
\[ \text{seqPerm} : \text{Seq} \times \text{Seq} \]
\[ \text{seqSwap} : \text{Seq} \times \text{int} \times \text{int} \rightarrow \text{Seq} \]
\[ \text{seqRemove} : \text{Seq} \times \text{int} \rightarrow \text{Seq} \]
\[ \text{seqNPermInv} : \text{Seq} \]

Fig. 5.1 The vocabulary \( \Sigma_{\text{seq}} \) of the theory \( T_{\text{seq}} \) of finite sequences

(The concrete, ASCII syntax is \( A::\text{seqGet} \), but we stick here with the slightly shorter notation \( \text{seqGet}_A \).) The constant \( \text{seqGetOutside} \) is an arbitrary element of the top type \( \text{Any} \). It is, e.g., used as the value of any attempt to access a sequence outside its range. \( \text{seqDef} \) is a variable binder symbol, check Section 2.3.1 for explanation. Its precise semantics is given in Definition 5.2 below. The reader may get a first intuition from the simple example \( \text{seqDef} \{ u \} (1, 5, u^2) \) that represents the sequence \( \langle 1, 4, 9, 16 \rangle \). We will comment on the symbols in the definitional extension, when we are finished with the core theory following page 152.

lenNonNegative
\[ \forall \text{Seq } s; (0 \leq \text{seqLen}(s)) \]

equalityToSeqGetAndSeqLen
\[ \forall \text{Seq } s_1, s_2; (s_1 = s_2) \rightarrow (\text{seqLen}(s_1) = \text{seqLen}(s_2)) \]
\[ \forall \text{int } i; (0 \leq i < \text{seqLen}(s_1)) \rightarrow (\text{seqGetAny}(s_1, i) = \text{seqGetAny}(s_2, i)) \]

getOfSeqDef
\[ \forall \text{int } i, r, l; \forall \text{Any } x; (\]
\[ ((0 \leq i \land i < r - l) \rightarrow (\text{seqGet}_A(\text{seqDef} \{ u \} (l, r, t), i) = \text{cast}_A(t\{ (l + i) / u \})) \LAND \]
\[ (r - l \leq i < r - l) \rightarrow (\text{seqGet}_A(\text{seqDef} \{ u \} (l, r, t), i) = \text{cast}_A(\text{seqGetOutside}))) \]

lenOfSeqDef
\[ \forall \text{int } r, l; \forall \text{Any } x; (\]
\[ ((0 \leq i \land i < r - l) \rightarrow (\text{seqLen}(\text{seqDef} \{ u \} (l, r, t)) = r - l \land \]
\[ (r - l \leq i \rightarrow (\text{seqLen}(\text{seqDef} \{ u \} (l, r, t)) = 0)) \]

Fig. 5.2 Axioms of the core theory \( \text{CoT}_{\text{seq}} \) (in mathematical notation)
5.2 Finite Sequences

The axioms of the core theory CoTseq are shown in Figure 5.2 in mathematical notation together with the names of the corresponding taclets. In getOfSeqDef the quantifier \( \forall \) binds the variables that may occur in term \( t \).

Definition 5.1 below extends the semantics of type domains given in Figure 2.10 on page 45. More precisely, the definition gives the construction to obtain \( D_{\text{Seq}}^{\#} \) when all other type domains are fixed.

**Definition 5.1 (The type domain \( D_{\text{Seq}}^{\#} \)).** The type domain \( D_{\text{Seq}}^{\#} \) is defined via the following induction:

\[
D_{\text{Seq}}^{\#} := \bigcup_{n \geq 0} D_{\text{Seq}}^{n}\]

where

\[
\begin{align*}
U & = D_{\text{Any}}^{\#} \setminus D_{\text{Seq}}^{\#} \\
D_{\text{Seq}}^{0} & = \{ \langle \rangle \} \\
D_{\text{Seq}}^{n+1} & = \{ \langle a_0, \ldots, a_{k-1} \rangle | k \in \mathbb{N} \text{ and } a_i \in D_{\text{Seq}}^{n} \cup U, 0 \leq i < k \} \quad \text{for } n \geq 0
\end{align*}
\]

The type domain for Seq being fixed we now may deliver on the forward reference after Definition 2.22 and define precisely the meaning of the variable binder symbol seqDef \{iv\}(le, ri, e) in the JFOL structure \( M \). As already done in Section 2.4.5 we will use the notation \( t_{M, \beta}^{\#} \) for term evaluation instead of \( \text{val}_{M, \beta}(t) \). We further will suppress \( \beta \) and write \( t_{M}^{\#} \) if it is not needed or not relevant.

**Definition 5.2.**

seqDef \{iv\}(le, ri, e)_{M, \beta}^{\#} =
\[
\begin{cases}
\langle a_0, \ldots, a_{k-1} \rangle & \text{if } (ri - le)^{\#, \beta} = k > 0 \text{ and } a_i = e^{\#, \beta_i} \\
\langle \rangle & \text{with } \beta_i = \beta[le + i/iv] \text{ and all } 0 \leq i < k \\
\end{cases}
\]

Remember, that \( \beta[le + i/iv] \) is the variable assignment that coincides with \( \beta \) except for the argument \( iv \) where it takes the value \( le + i \).

The core vocabulary of CoTseq is interpreted as follows:

**Definition 5.3.**

1. seqGet\_{\#}^{\#}(\langle a_0, \ldots, a_{n-1} \rangle, i) =
\[
\begin{cases}
\text{cast}_{\#}^{\#}(a_i) & \text{if } 0 \leq i < n \\
\text{cast}_{\#}(\text{seqGetOutside}^{\#}) & \text{otherwise}
\end{cases}
\]
2. seqLen\_{\#}^{\#}(\langle a_0, \ldots, a_{n-1} \rangle) = n
3. seqGetOutside\_{\#}^{\#} \in D_{\text{Any}}^{\#} \text{ arbitrary.}

To have a name for it we might call a structure \( M \) in the vocabulary \( \Sigma_f \) (see Figure 2.4) plus the core vocabulary of finite sequences a CoreSeq structure, if its restriction to the JFOL vocabulary is a JFOL structure as defined in Section 2.4.5 and, in addition \( M \) satisfies Definition 5.3. We observe, that the expansion of a JFOL structure \( M_0 \) to a CoreSeq structure is uniquely determined once an interpretation seqGetOutside\_{\#}^{\#} is chosen.
Theorem 5.4. The theory $\text{CoT}_{\text{seq}}$ is consistent.

Proof. It is easily checked that the axioms in Figure 5.2 are true in all $\text{CoreSeq}$ structures. The explicit construction guarantees that there is at least one $\text{CoreSeq}$ structure.

$$\forall s \in \text{Seq} \; \forall i : (0 \leq i < \text{seqLen}(s) \rightarrow \neg \text{instanceSeq}(\text{seqGet}_{\text{seq}}(s, i))) \rightarrow \text{seqDepth}(s) \leq 0 \land$$

$$\forall s \in \text{Seq} \; \forall i : (0 \leq i < \text{seqLen}(s) \land \text{instanceSeq}(\text{seqGet}_{\text{seq}}(s, i))) ightarrow \text{seqDepth}(s) > \text{seqDepth}(\text{seqGet}_{\text{seq}}(s, i)) \land$$

$$\forall s \in \text{Seq} \; \exists i : (0 \leq i < \text{seqLen}(s) \land \text{instanceSeq}(\text{seqGet}_{\text{seq}}(s, i))) ightarrow \exists i : (0 \leq i < \text{seqLen}(s) \land \text{instanceSeq}(\text{seqGet}_{\text{seq}}(s, i)) \land$$

$$\text{seqDepth}(s) = \text{seqDepth}(\text{seqGet}_{\text{seq}}(s, i)) + 1)$$

Fig. 5.3 Definition of seqDepth

We observe that $\text{seqDepth}(s)$ as defined in Figure 5.3 equals the recursive definition

$$\text{seqDepth}(s) = \max \{ \text{seqDepth}(\text{seqGet}_{\text{seq}}(s, i)) \mid 0 \leq i < \text{seqLen}(s) \land \text{instanceSeq}(\text{seqGet}_{\text{seq}}(s, i)) \}$$

with the understanding that the maximum of the empty set is 0. Since we have not introduced the maximum operator we had to resort to the formula given above. The function seqDepth is foremost of theoretical interest and at the moment of this writing not realized in the KeY system. seqDepth is an integer denoting the nesting depth of sequence $s$. If $s$ has no entries that are themselves sequences then seqDepth $s = 0$. For a sequence $s_{\text{int}}$ of sequences of integers we would have seqDepth $s_{\text{int}} = 0$.

In Figure 5.4 the mathematical formulas defining the remaining noncore vocabulary are accompanied by the names of the corresponding tactics. A few explaining comments will help the reader to grasp their meaning. The subsequence seqSub$(s, i, j)$ from $i$ to $j$ of sequence $s$ includes the $i$-th entry, but excludes the $j$-th entry. In the case $¬(i < j)$ it will be the empty sequence, this is a consequence of the semantics of seqDef. The term seqIndexOf$(s, t)$ denotes the least index $n$ such that seqGet$_{\text{seq}}(s, n) = t$ if there is one, and is undefined otherwise. See Section 2.3.2 on how undefinedness is handled in our logic. A sequence $s$ satisfies the predicate seqNPerm$(s)$ if it is a permutation of the integers $\{ 0, \ldots, \text{seqLen}(s) - 1 \}$. The binary predicate seqPerm$(s_1, s_2)$ is true if $s_2$ is a permutation of $s_1$. Thus seqPerm$(\{ 5, 4, 0, 2, 3, 1 \})$ and seqPerm$(\{ a, b, c \}, \{ b, a, c \})$ are true.

Careful observation reveals that the interpretation of the vocabulary outside the core vocabulary is uniquely determined by the definitions in Figures 5.3 and 5.4.

We establish the following notation:

Definition 5.5. By $T_{\text{seq}}$ we denote the theory given by the core axioms $\text{CoT}_{\text{seq}}$ plus the definitions from Figures 5.3 and 5.4.

On the semantic side we call a structure $\mathcal{M}$ in the vocabulary $\Sigma_I$ plus $\Sigma_{\text{seq}}$ a Seq structure if the restriction of $\mathcal{M}$ to $\Sigma_I$ is a JFOL structure and $\mathcal{M}$ satisfies Definitions 5.3 and 5.4.
\begin{align*}
\text{defOfEmpty} & \equiv \text{seqDef}\{iv\}(0,0,x) \\
& x \text{ is an arbitrary term of type Any not containing the variable iv.}
\end{align*}

\begin{align*}
\text{defOfSingleton} & \forall x; (\text{seqSingleton}(x) \equiv \text{seqDef}\{iv\}(0,1,x)) \\
\text{defOfConcat} & \forall s_1, s_2; (\text{seqConcat}(s_1, s_2) \equiv \text{seqDef}\{iv\}(0,\text{seqLen}(s_1) + \text{seqLen}(s_2), \text{if } iv < \text{seqLen}(s_1) \text{ then seqGet}_{\text{Any}}(s_1, iv) \text{ else seqGet}_{\text{Any}}(s_2, iv - \text{seqLen}(s_1))))
\end{align*}

\begin{align*}
\text{defOfSub} & \forall s; \forall i, j; (\text{seqSub}(s, i, j) \equiv \text{seqDef}\{iv\}(i, j, \text{seqGet}_{\text{Any}}(s, iv))) \\
\text{defOfReverse} & \forall s; (\text{seqReverse}(s) \equiv \text{seqDef}\{iv\}(0, \text{seqLen}(s), \text{seqGet}_{\text{Any}}(s, \text{seqLen}(s) - iv - 1))) \\
\text{seqIndexOf} & \forall s; \forall t; \forall n; (0 \leq n < \text{seqLen}(s) \land \text{seqGet}_{\text{Any}}(s, n) \equiv t \land \\
& \forall m; (0 \leq m < n \rightarrow \text{seqGet}_{\text{Any}}(s, m) \neq t) \\
& \rightarrow \text{seqIndexOf}(s, t) \equiv n)
\end{align*}

\begin{align*}
\text{seqNPPermDefReplace} & \forall s; (\text{seqNPPerm}(s) \leftrightarrow \\
& \forall t; (0 \leq i < \text{seqLen}(s) \rightarrow \exists j; (0 \leq j < \text{seqLen}(s) \land \text{seqGet}_{\text{Any}}(s, j) \equiv t))) \\
\text{seqPermDef} & \forall s_1, s_2; (\text{seqPerm}(s_1, s_2) \equiv \text{seqLen}(s_1) \equiv \text{seqLen}(s_2) \land \\
& \exists s; (\text{seqLen}(s) \equiv \text{seqLen}(s_1) \land \text{seqNPPerm}(s) \land \\
& \forall t; (0 \leq i < \text{seqLen}(s) \\
& \rightarrow \text{seqGet}_{\text{Any}}(s_1, i) \equiv \text{seqGet}_{\text{Any}}(s_2, \text{seqGet}_{\text{Any}}(s, i)))))
\end{align*}

\begin{align*}
\text{defOfSwap} & \forall s; \forall t; (\text{seqSwap}(s, i, j) \equiv \\
& \text{seqDef}\{iv\}(0, \text{seqLen}(s), \text{if } -(0 \leq i < \text{seqLen}(s) \land 0 \leq j < \text{seqLen}(s) \text{ then seqGet}_{\text{Any}}(s, iv) \text{ else if } iv \equiv i \text{ then seqGet}_{\text{Any}}(s, j) \text{ else if } iv \equiv j \\
& \text{ then seqGet}_{\text{Any}}(s, i) \text{ else seqGet}_{\text{Any}}(s, iv))))
\end{align*}

\begin{align*}
\text{defOfRemove} & \forall s; \forall t; (\text{seqRemove}(s, i) \equiv \text{if } i < 0 \lor \text{seqLen}(s) \leq i \text{ then } s \\
& \text{else } \text{seqDef}\{iv\}(0, \text{seqLen}(s) - 1, \text{if } iv < i \text{ then seqGet}_{\text{Any}}(s, iv) \text{ else seqGet}_{\text{Any}}(s, iv + 1))))
\end{align*}

\begin{align*}
\text{defOfNPPermInv} & \forall s; (\text{seqNPPermInv}(s) \equiv \text{seqDef}\{iv\}(0, \text{seqLen}(s), \text{seqIndexOf}(s, iv))))
\end{align*}

\textbf{Fig. 5.4} Definition for noncore vocabulary in mathematical notation
Theorem 5.6. The theory $T_{\text{Seq}}$ is consistent.

Proof. The consistency of $T_{\text{Seq}}$ follows from the consistency of $\text{CoT}_{\text{Seq}}$ since it is a definitional extension.

A proof of Theorem 5.6 together with a detailed review of the concept of definitional extensions, plus statement and proof of the relative completeness of $T_{\text{Seq}}$ can be found in the technical report on first-order logic available from the companion website to this book www.key-project.org/thebook2.

1 seqSelfDefinition
$$\forall \text{Seq } s; (s \equiv \text{seqDef } \{u\}(0, \text{seqLen}(s), \text{seqGet}_\text{Seq}(s, u)))$$

2 seqOutsideValue
$$\forall \text{Seq } s; (?\text{int } i; ((i < 0 \lor \text{seqLen}(s) \leq i) \rightarrow \text{seqGet}_\text{Seq}(s, i) \equiv (\alpha)\text{seqGetOutside}))$$

3 castedGetAny
$$\forall \text{Seq } s; (?\text{int } i; ((\beta)\text{seqGet}_\text{Seq}(s, i) \equiv \text{seqGet}_\text{Seq}(s, i)))$$

4 getOfSeqSingleton
$$\forall \text{Any } x; (?\text{int } i; (\text{seqGet}_\text{Seq}(\text{seqSingleton}(x), i) \equiv i \neq 0 \times (\alpha)x \times (\alpha)\text{seqGetOutside}))$$

5 getOfSeqConcat
$$\forall \text{Seq } s, s_2; (?\text{int } i; (\text{seqGet}_\text{Seq}(\text{seqConcat}(s, s_2), i) \equiv \text{if } i < \text{seqLen}(s)$$
$$\text{then } \text{seqGet}_\text{Seq}(s, i)$$
$$\text{else } \text{seqGet}_\text{Seq}(s_2, i - \text{seqLen}(s))))$$

6 getOfSeqSub
$$\forall \text{Seq } s, \text{int from}, to, i; (\text{seqGet}_\text{Seq}(\text{seqSub}(s, \text{from}, to), i) \equiv \text{if } 0 \leq i \land i < (\text{to} - \text{from})$$
$$\text{then } \text{seqGet}_\text{Seq}(s, i + \text{from})$$
$$\text{else } (\alpha)\text{seqGetOutside}))$$

7 getOfSeqReverse
$$\forall \text{Seq } s; (?\text{int from}, to, i; (\text{seqGet}_\text{Seq}(\text{seqReverse}(s), i) \equiv \text{seqGet}_\text{Seq}(s, \text{seqLen}(s) - 1 - i))$$

8 lenOfSeqEmpty
$$\text{seqLen}(\text{seqEmpty}) = 0$$

9 lenOfSeqSingleton
$$\forall \text{Any } x; (\text{seqLen}(\text{seqSingleton}(x)) = 1)$$

10 lenOfSeqConcat
$$\forall \text{Seq } s, s_2; (\text{seqLen}(\text{seqConcat}(s, s_2)) \equiv \text{seqLen}(s) + \text{seqLen}(s_2))$$

11 lenOfSeqSub
$$\forall \text{Seq } s; (?\text{int from}, to; \text{seqLen}(\text{seqSub}(s, \text{from}, to)) \equiv \text{if } \text{from} < \text{to} \times \text{then } (\text{to} - \text{from}) \times \text{else } 0$$

12 lenOfSeqReverse
$$\forall \text{Seq } s; (\text{seqLen}(\text{seqReverse}(s)) \equiv \text{seqLen}(s))$$

13 seqConcatWithSeqEmpty
$$\forall \text{Seq } s; (\text{seqConcat}(s, \text{seqEmpty}) \equiv s)$$

14 seqReverseOfSeqEmpty
$$\text{seqReverse}(\text{seqEmpty}) = \text{seqEmpty}$$

Fig. 5.5 Some derived rules for finite sequences

Figure 5.5 lists some consequences that can be derived from the definitions in Figure 5.4 and the Core Theory. The entry 1 is a technical lemma that is useful
in the derivation of the following lemmas in the list. The entry 2 clarifies the role of the default value \textit{seqGetOutside}; it is the default or error value for any out-of-range access. Rules 2 to 7 are schematic rules. These rules are applicable for any instantiations of the schema variable \( \alpha \) by a type. Entry 3 addresses an important issue: on one hand there is the family of function symbols \textit{seqGet}\( _\alpha \), on the other hand there are the cast expressions \((\alpha)\textit{seqGet}\textit{Any}\). The lemma says that both coincide. The entries 4 to 12 allow to determine the access function and the length of the empty sequence, singleton, concatenation, subsequence and reverse constructors. The last two entries 13 and 14 are examples for a whole set of rules that cover corner cases of the constructors involved.

\begin{verbatim}
1 seqNPermRange
   \forall Seq s; (seqNPerm(s) \rightarrow
   \forall int i; (0 \leq i \land i < seqLen(s) \rightarrow 0 \leq seqGet\textit{int}(s, i) \land seqGet\textit{int}(s, i) < seqLen(s))])

2 seqNPermInjective
   \forall Seq s; (seqNPerm(s) \land
   \forall int i, j; (0 \leq i \land i < seqLen(s) \land 0 \leq j \land j < seqLen(s) \land seqGet\textit{int}(s, i) = seqGet\textit{int}(s, j))
   \rightarrow i = j)

3 seqNPermEmpty
   seqNPerm(seqEmpty)

4 seqNPermSingleton
   \forall int i; (seqNPerm(seqSingleton(i)) \leftrightarrow i = 0)

5 seqNPermComp
   \forall Seq s1, s2; (seqNPerm(s1) \land seqNPerm(s2) \land seqLen(s1) = seqLen(s2) \rightarrow
   seqNPerm(seqDef\{u\}(0, seqLen(s1), seqGet\textit{int}(s1, seqGet\textit{int}(s2, u)))))

6 seqPermTrans
   \forall Seq s1, s2, s3; (seqPerm(s1, s2) \land seqPerm(s2, s3) \rightarrow seqPerm(s1, s3))

7 seqPermRefl
   \forall Seq s; (seqPerm(s, s))
\end{verbatim}

\textbf{Fig. 5.6} Some derived rules for permutations

Figure 5.6 lists some derived rules for the one-place predicate \textit{seqNPerm} and the two-place predicate \textit{seqPerm} that follow from the definitions in Figure 5.4. Surprisingly, none of the proofs apart from the one for \textit{seqNPermRange} needs induction. This is mainly due to the presence of the \textit{seqDef\{\}}(,) construct. The lemma \textit{seqNPermRange} itself is a kind of pigeon-hole principle and could only be proved via induction.

Applications of the theory of finite sequences can be found in Section 16.5 and foremost in Chapter 19.
5.3 Strings

Java strings are implemented as objects of their own and they are not identified with arrays of characters. This eases treatment of strings in our program logic as we can reuse all the mechanisms already in place for objects. So, do we need special treatment for them at all? Why not simply use contracts as for other API classes?

To answer the first question: Although strings are normal objects, the Java Language Specification [Gosling et al., 2013] provides some additional infrastructure not available for other kinds of objects. In particular, the existence of string literals like "Hello, world" requires additional thought. A string literal is a reference to a string object whose content coincides with the string literal name within the quotation marks. The problem to be solved is to make string literals 'behave' like integer or Boolean literals. For instance, the expression "Hello" == "Hello" should always be true. To solve this issue, Java ensures that all occurrences of the same literal reference the same object. To ensure this behavior Java manages a pool of strings in which all strings referenced by string literals (actually, all compile time constants of type String) are put. Nevertheless, the taken solution does not hide completely that string literals are different from other literals, e.g., the expression new String("Hello") == "Hello" evaluates to false. To represent the pool, we could model it in Java itself. This solution would allow us to be mostly ignorant to strings on the logic level, but introduce a lot of clutter in the reasoning process when string literals are involved. Instead we use an alternative route and model the pool purely on the logic level, which allows us a more streamlined representation and deemphasizes the use of a pool.

We go a step further and introduce a kind of "ghost" field for Java string objects that assigns each string a finite sequence of characters based on the Sequence data type introduce in the previous section. As specifications about strings express properties about their content, e.g., that the content equals or matches a given expression (e.g., a regular expression), providing an abstract data type for strings allows us to separate concerns and to ease writing of specifications.

In this section we describe three parts that constitute our handling of Java strings: The theory $T_{cl}$ of sequences of characters representing the content of a Java string as an extension of the theory of finite sequences, a theory $T_{rex}$ of regular expressions to express and reason conveniently about the content of strings, and finally we conclude with the actual theory $T_{java.lang.String}$ of Java strings, which uses the previous two theories to model Java strings.

5.3.1 Sequences of Characters

Characters are not represented as a distinct type, but as integers which are interpreted as the characters unicode (UTF-16) representation. A finite sequence of characters is used to model the underlying theory to support Java Strings. The length of the sequence is the number of 16-bit unicode values. This value might not coincide with
the number of actual characters as some unicode characters need to be represented by
two 16-bit numbers. The presentation below and the actual implementation assumes
that only unicode characters in the range 0x0000 – 0xFFFF are used.

We extend the theory of finite sequences by the additional functions and predicates shown in Table 5.1. They allow us later to specify the behavior of the String’s
method concisely. The first four functions return the first (or last) index of a sequence

Table 5.1  The additional functions and predicates of $T_{cl}$ (int$^*$ is used to indicate that a character’s
UTF-16 unicode representation is expected as argument; the actual type is int)

<table>
<thead>
<tr>
<th>Extensions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>clIndexOfChar : Seq $\times$ int $\times$ int $\rightarrow$ int</td>
<td></td>
</tr>
<tr>
<td>clIndexOfCl  : Seq $\times$ int $\times$ Seq $\rightarrow$ int</td>
<td></td>
</tr>
<tr>
<td>clLastIndexOfChar : Seq $\times$ int $\times$ int $\rightarrow$ int</td>
<td></td>
</tr>
<tr>
<td>clLastIndexOfCl  : Seq $\times$ int $\times$ Seq $\rightarrow$ int</td>
<td></td>
</tr>
<tr>
<td>clReplace : Seq $\times$ int$^<em>$ $\times$ int$^</em>$ $\rightarrow$ Seq</td>
<td></td>
</tr>
<tr>
<td>clTranslateInt : int $\rightarrow$ Seq</td>
<td></td>
</tr>
<tr>
<td>clRemoveZeros : Seq $\rightarrow$ Seq</td>
<td></td>
</tr>
<tr>
<td>clHashCode : Seq $\rightarrow$ int</td>
<td></td>
</tr>
<tr>
<td>clStartsWith : Seq $\times$ Seq</td>
<td></td>
</tr>
<tr>
<td>clEndsWith : Seq $\times$ Seq</td>
<td></td>
</tr>
<tr>
<td>clContains : Seq $\times$ Seq</td>
<td></td>
</tr>
</tbody>
</table>

starting from the position given as third argument at which to find the specified
character (start of the given character sequence) or $–1$ if no such character (character
sequence) exists (this differs from seqIndexOf, which is undefined for elements not
occurring in a sequent). Function clReplace($s, c_1, c_2$) evaluates to a sequence equal
to $s$ except that all occurrences of character $c_1$ have been replaced by character $c_2$.
Function clTranslateInt takes a character sequence specifying a number and translates it into the corresponding integer. It comes paired with the auxiliary function
clRemoveZero which removes any leading zeros, as in "000123", first, before the
result can be handed over to clTranslateInt in order to rewrite it into the integer 123.

The predicates clContains, clStartsWith, and clEndsWith evaluate to true if the
character sequence given as first argument contains, starts or ends with the character
sequence given as second argument, respectively.

The actual axiomatizations are rather technical, but not complicated. Those
for clIndexOfChar and clContains are shown in Figure 5.7. The axiomatization
of clIndexOfChar makes use of JavaDL’s ifEx operator to determine the minimal
(first) index of the searched character, if one exists.

For convenience reasons, we write short ”abc” instead of the actual term
seqConcat(seqSingleton(′a′), seqConcat(seqSingleton(′b′), seqSingleton(′c′))). The
pretty printer of KeY outputs a character sequence in this way (outside of modalities),
and the parser accepts string literals as an alternative syntax for character sequences.
indexOf
∀ Seq l, c; ∀ int i;
   (clIndexOfChar(l, c, i) =
    ifEx int iv; (i ≥ 0 ∧ iv ≥ i ∧ iv < seqLen(l) ∧ seqGet(l, iv) = c)
    then(iv) else(−1))

containsAxiom
∀ Seq textString, searchString;
   (clContains(textString, searchString) ↔
    ∃ int iv; (iv ≥ 0 ∧ iv + seqLen(searchString) ≥ seqLen(textString)
    ∧ seqSub(textString, iv, iv + seqLen(searchString)) = searchString))

Fig. 5.7 Axioms for clIndexOfChar and clContains

5.3.2 Regular Expressions for Sequences

To be able to conveniently specify methods manipulating strings, the theory T\textsubscript{rex} allows one to match elements of type Seq using regular expression. Pattern expressions (PExp) are represented as terms of type rex. Table 5.2 lists the PExp constructors. For instance, the pattern represented by the term repeatStar( rex("ab") ) matches a finite but arbitrary repetition of the word "ab". Match expressions are constructed using the predicate match(rex, Seq). The predicate match takes two arguments: a PExp as first argument and the concrete character sequence to be matched against the pattern as second argument. The match expression is true if and only if the provided pattern matches the complete Seq.

Our calculus features a complete axiomatization of the pattern and matching language. Further, there are a number of derived rules to reduce and simplify pattern and match expression terms as far as possible. We give here only a few typical representatives of these axioms and rules.

The first axiom maps the alternative pattern constructor back to a logical disjunction:

altAxiom
∀ rex pe1, pe2; ∀ Seq cl;
   match(alt(pe1, pe2), cl) ↔ (match(pe1, cl) ∨ match(pe2, cl))

\[\text{Remark: } T_{\text{rex}} \text{ goes actually beyond regular expressions.}\]
The second axiom removes the pattern concatenation by guessing the index where to split the text to be matched into two parts. Each part is then independently matched against the corresponding subpattern:

\[
\text{regConcatAxiom} \quad \forall \text{rex} \; \pe_1, \pe_2; \forall \text{Seq} \; 
\text{cl} ; (\text{match} (\text{regConcat} (\pe_1, \pe_2), \text{cl}) \leftrightarrow (\exists \text{i} . (i \geq 0 \land i \leq \text{seqLen} (\text{cl}) \land \text{match} (\pe_1, \text{seqSub} (\text{cl}, 0, i)) \land \text{match} (\pe_2, \text{seqSub} (\text{cl}, i, \text{seqLen} (\text{cl}))))))
\]

A typical reduction rule aiming to reduce the complexity of is for instance:

\[
\text{regConcatConcreteStringLeft} \quad \exists \text{rex} \; \pre; \exists \text{Seq} \; s, \text{cl}; (\text{match} (\text{regConcat} (\text{rex}(s), \pe), \text{cl}) \leftrightarrow (\text{seqLen}(s) \leq \text{seqLen(\text{cl})} \land \text{match} (\text{rex}(s), \text{seqSub}(\text{cl}, 0, \text{seqLen}(s))) \land \text{match} (\pe, \text{seqSub}(\text{cl}, \text{seqLen}(s), \text{seqLen}(\text{cl})))))
\]

### 5.3.3 Relating Java String Objects to Sequences of Characters

The previous sections introduced the logic representation of character sequences and regular expressions. To achieve our goal to specify and verify programs in presence of Strings, the abstract representation of a string’s content and the implementation of Java’s \texttt{String} class need to be related.

This could be simply achieved by introducing a ghost field and keeping the content on the heap. We choose a similar but slightly different modeling, which simplifies verification down the road by exploiting that once a String instance is created, its content does not change. This is the same situation as for the length field of an array, and we use the same idea. The function \textit{strContent} : \texttt{java.lang.String} \rightarrow \texttt{Seq} maps each String instance to a sequence of characters. It is left unspecified initially and upon creation of a new String instance \textit{s} representing, e.g., the character list \textit{sc}, the formula \textit{strContent}(\textit{s}) = \textit{sc} is added to the antecedent of the formula. This is well-defined as the content of a String instance cannot be changed. The following sequent calculus rule illustrates this mechanism:

\[
\text{stringConcat} \quad \Gamma, \text{strContent}(sk) \vdash \text{seqConcat}(\text{strContent}(s_1), \text{strContent}(s_2)), sk \neq \text{null} \Longrightarrow \\
\{ v := sk \} \{ \text{heap} := \text{create(\text{heap}, sk)} \} (\pi \omega) \phi, \Delta \\
\Gamma \Longrightarrow (\pi \; v = s_1 + s_2; \; \omega) \phi, \Delta
\]

Schema variables \(v, s_1, s_2\) match local program variables of type \texttt{java.lang.String} and \(sk\) is a fresh Skolem-constant.
5.3.4 String Literals and the String Pool

The Java string pool caches String instances using their content as key. On start-up of the virtual machine and after class loading all compile-time constant of type String (in particular all string literals) are resolved to an actual String object. New elements can be added to the cache at run-time with method `intern()`, but Java programs cannot remove elements from the cache.

We model the string pool as an injective function

$$strPool : Seq \rightarrow \text{java.lang.String}$$

The assignment rule for string literals in the presence of the string pool can now be defined as follows:

$$\text{stringAssignment}$$

$$\Gamma, \text{strContent} \leftarrow \text{strPool}(sLit_CL) = sLit_CL,$$

$$\text{strPool}(sLit_CL) \neq \text{null},$$

$$\text{select boolean}(\text{heap}, \text{strPool}(\text{strContent}(sLit_CL)), \text{created}) \equiv \text{TRUE}$$

$$\implies \{ v := \text{strPool}(sLit_CL) \} \langle \pi \omega \rangle \phi, \Delta$$

Here $sLit$ is a schema variable matching string literals and $sLit_CL$ denotes the finite sequence $Seq$ representation of the matched string literal $sLit$.

One side remark at this point concerning the concatenation of string literals, i.e., how a program fragment of the kind $v = "a" + "b"$; is treated. In this case the expression $"a" + "b"$ is a compile time constant, which are as their name suggests evaluated at compile time. Hence, any such expression has already been replaced by the result $"ab$ when reading in the program (in other words, in JavaDL all compile time constants are already replaced by their fully evaluated literal expression).

Finally, we give one of the rules for updating the Java string pool with a new element. Note, this rule is actually specified as a contract of method `intern()` of class String:

$$\text{updatePool}$$

$$\Gamma, \neg (v \equiv \text{null}), \text{strPool}(\text{strContent}(v)) \neq \text{null},$$

$$\text{select boolean}(\text{heap}, \text{strPool}(\text{strContent}(v)), \text{created}) \equiv \text{TRUE}$$

$$\implies \{ r := \text{strPool}(\text{strContent}(v)) \} \langle \pi \omega \rangle \phi, \Delta$$

$$\Gamma, \neg (v \equiv \text{null}) \implies \langle \pi r = .\text{intern}(); \omega \rangle \phi, \Delta$$
5.3.5 *Specification of the Java String API*

To obtain a complete calculus for Java strings, additional rules have to be created which translate an integer or the null reference to its String representation. The formalization of the necessary translate functions is rather tedious, but otherwise straightforward. The technical details are described in [Geilmann, 2009].

Based on the formalization described in this section, we specified the majority of the methods declared and implemented in the java.lang.String class. The Seq ADT functions have been chosen to represent closely the core functionality provided by the String class. The specification of the methods required then merely to consider the border cases of most of the methods. Border cases are typically those cases where the ADT has been left underspecified and that cause an exception in Java.

5.4 **Integers**

Arithmetic reasoning is required for a number of verification tasks like proving that a certain index is within the array’s bounds, that no arithmetic exception occurs and, of course, that a method computes the correct value. Mathematical integers have already been covered in Section 2.4.5, in this section we highlight the axiomatization of integers with respect to their finite integral counterparts used in Java.

In lieu of the whole numbers \( \mathbb{Z} \), programming languages usually use finite integral types based on a two-complement representation. For instance, Java’s integral types (byte, char, short, int and long) are represented in 8-bit, 16-bit, 32-bit and 64-bit two-complement representation (with the exception of char which is represented as an unsigned 16-bit number).

The finiteness of integral types and the often used modulo arithmetics entail the possibility of underflows and overflows. While sometimes intended, they are also a source of bugs leading to unexpected behavior. As pointed out by Joshua Bloch\(^2\) most binary search algorithms are broken because of an overflow that might happen when computing the middle of the interval \( \text{lower} \ldots \text{upper} \) by \( (\text{upper}+\text{lower})/2 \) (e.g., for \( \text{lower} \) equal to 100 and \( \text{upper} \) equal to the maximal value of its integral type).

The question arises: How do we model finite integral types within the program logic. One possibility is to define new sorts to model the required fixed-precision numbers together with functions for addition and subtraction, e.g., as a general fixed-width bit-vector theory. Another approach, and that is the one we pursued in JFOL, is to map all Java arithmetic operations to standard arithmetic operations without the need to introduce new additional sorts.

\(^2\) googleresearch.blogspot.se/2006/06/extra-extra-read-all-about-it-nearly.html
5.4.1 Core Integer Theory

The predefined sort \texttt{int} is evaluated to the set of whole numbers \( \mathbb{Z} \). Table 5.3 shows a selection of the most important interpreted core functions. All of them are interpreted canonically. In case of division, \texttt{div} rounding towards the nearer lower number takes place, and \texttt{mod} is interpreted as the modulo function.

On top of these core functions, derived functions (shown in Table 5.3 as extensions) are defined. Their domain is still the whole numbers, i.e., no modulo arithmetics are involved, but otherwise they reflect the division and modulo semantics in Java more closely. Namely, function \texttt{jdiv} is interpreted as the division on \( \mathbb{Z} \) rounding towards zero, while \texttt{jmod} is interpreted as the remainder function in opposite to the modulo function.

Finally, functions like \texttt{addJint}, \texttt{addJlong} etc. are defined in such a way that they reflect the modulo semantics of the Java operations. They are axiomatized solely using the functions shown in Table 5.3

<table>
<thead>
<tr>
<th>Table 5.3</th>
<th>Core and extension functions for the \texttt{int} data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>'+', addition on ( \mathbb{Z} )</td>
</tr>
<tr>
<td>sub</td>
<td>'-', subtraction on ( \mathbb{Z} )</td>
</tr>
<tr>
<td>div</td>
<td>'/', division on ( \mathbb{Z} ) (Euclidean semantics)</td>
</tr>
<tr>
<td>mod</td>
<td>'%', modulo on ( \mathbb{Z} )</td>
</tr>
<tr>
<td>mul</td>
<td>'*', multiplication on ( \mathbb{Z} )</td>
</tr>
<tr>
<td>Extensions</td>
<td></td>
</tr>
<tr>
<td>jdiv</td>
<td>n/a, division on ( \mathbb{Z} ) (rounding towards zero)</td>
</tr>
<tr>
<td>jmod</td>
<td>n/a, remainder on ( \mathbb{Z} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.4</th>
<th>Functions for Java arithmetics</th>
</tr>
</thead>
<tbody>
<tr>
<td>addJint/addJlong</td>
<td>addition with overflow for \texttt{int}/\texttt{long}</td>
</tr>
<tr>
<td>divJint/divJlong</td>
<td>modulo with overflow for \texttt{int}/\texttt{long}</td>
</tr>
<tr>
<td>moduloByte/moduloChar/moduloShort/...</td>
<td>modulo operation mapping numbers into their respective range</td>
</tr>
</tbody>
</table>

On the calculus side the axiom for e.g. \texttt{addJlong} is given as rewrite rule

\[
\text{addJlong}(fst, snd) \rightsquigarrow \text{moduloLong}(\text{add}(fst, snd))
\]

and expresses the meaning of addition with overflow w.r.t. the value range of \texttt{long} in terms of the standard arithmetic addition and a modulo operation. The calculus rewrite rule defining function \texttt{moduloLong} is

\[
\text{moduloLong}(i) \rightsquigarrow \\
\text{add}(\text{long}_\text{MIN}, \text{mod}(\text{add}(\text{long}_\text{HALFRANGE}, i), \text{long}_\text{RANGE}))
\]
5.4. Integers

Its definition refers only to the standard arithmetic functions for addition and modulo. The only other elements \texttt{long\_MIN}, \texttt{long\_HALFRANGE} and \texttt{long\_RANGE} are constants like the smallest number of type \texttt{long} or the cardinality of the set of all numbers in \texttt{long}.

In our logic all function symbols are interpreted as total functions. There are several possibilities to deal with terms like $\texttt{div}(x, 0)$ like (a) returning a default value, (b) returning a special named error element or (c) using underspecification. Solution (a) might easily hide an existing problem in the specification as it might render it provable but not matching the specifier's intuition, solution (b) would require to extend the definition of all functions defined on the integers to deal with the special error element. For these reasons, we choose underspecification, i.e., the semantics of logic does not fix a specific value for $\texttt{div}$ in case of a division by zero. Instead each JFOL structure $\mathcal{M}$ assigns $\texttt{div}^M(d, 0)$ a fixed but unknown integer value for each dividend $d \in \mathbb{Z}$. These values may differ between different Kripke structures but are not state-dependent in the sense that $\texttt{div}(1, 0)$ is assigned a different value in different states.

Besides the avoidance of a proliferation of special error cases or dealing with partial functions, the use of underspecification in the sketched manner provides additional advantages, e.g., a formula like

- $\texttt{div}(x, 0) \neq \texttt{div}(x + 1, 0)$ is neither a tautology nor unsatisfiable, but
- $\texttt{div}(x, 0) \neq \texttt{div}(x, 0)$ remains a tautology.

5.4.2 Variable Binding Integer Operations

Variable binding operators are used to express sums and products. Our theory of integers supports the general versions $\texttt{sum}(T \ x)(\phi(x), t)$, $\texttt{prod}(T \ x)(\phi(x), t)$ as well as their bounded variants $\texttt{bsum}(\texttt{int} \ x)(\texttt{start}, \texttt{end}, t)$ and $\texttt{bprod}(\texttt{int} \ x)(\texttt{start}, \texttt{end}, t)$. These functions are defined as follows:

\[
\texttt{sum}(T \ x)(\phi(x), t)^\# = \sum_{\phi(x)^\#} r^\# \quad \text{prod}(T \ x)(\phi(x), t)^\# = \prod_{\phi(x)^\#} r^\#
\]

\[
\texttt{bsum}(\texttt{int} \ x)(\texttt{start}, \texttt{end}, t)^\# = \sum_{x=\texttt{start}^\#}^{\texttt{end}^\# - 1} r^\# \quad (\text{if } \texttt{start} > \texttt{end} , \text{otherwise } 0)
\]

\[
\texttt{bprod}(\texttt{int} \ x)(\texttt{start}, \texttt{end}, t)^\# = \prod_{x=\texttt{start}^\#}^{\texttt{end}^\# - 1} r^\# \quad (\text{if } \texttt{start} > \texttt{end} , \text{otherwise } 1)
\]

The logic axiomatization for the bounded sum is given as: For any int-typed term $t$

\[
\forall \texttt{int } start, end: (\texttt{bsum}(\texttt{int} \ x)(\texttt{start}, \texttt{end}, t) = \text{if}(\texttt{start} < \texttt{end}) \text{then } \texttt{bsum}(\texttt{int} \ x)(\texttt{start}, \texttt{end} - 1, t) + (\{\texttt{\_subst } x; \texttt{start}\} t) \text{else}(0))
\]
The bounded sum and bounded product are inclusive for the first argument \textit{start} and exclusive for the second argument \textit{end}. Besides the axioms there are as usual a large number of lemmas that ease reasoning and increase automation. We show here the Taclet definition for the splitting a bounded sum as it highlights a feature of Taclet language that allows one to specify triggers:

\begin{verbatim}
KeY

bsum_split { 
  \schemaVar \term int \variables int x; 
  \find(bsum{x;}(low, high, t)) 
  \varcond( \notFreeIn(x, low), \notFreeIn(x, middle), \notFreeIn(x, high) ) 
  \replacewith( 
    \if(low <= middle & middle <= high) 
      bsum{x;}(low, middle, t) + bsum{x;}(middle,high,t) 
    \else(bsum{x;}(low, high, t)) )
  \heuristics(comprehension_split, triggered)
  \trigger{middle} bsum{x;}(low, middle, t)
  \avoid middle <= low, middle >= high;
};
\end{verbatim}

The above rule splits the bounded sum expression somewhere between the lower and upper bound. The trigger specification is used by the strategies to determine good candidates for the splitting point. A trigger consists of three parts: (i) the schema variables to be instantiated by the trigger (trigger variables) (here: \texttt{middle}); (ii) the pattern to be matched against an existing term (or formula) in the sequent (here: \texttt{bsum{x;}(low, middle, t)}) (the match determines the value of the trigger variables) and (iii) an optional part \texttt{avoid} that specifies the condition under which a candidate should be rejected. In the above case the trigger searches for positions at which to split the bounded sum into two parts such that the first summand already occurs in the sequent. The optional avoid part prevents superfluous splits outside the bounds (or directly at the start).

\subsection{5.4.3 Symbolic Execution of Integer Expressions in Programs}

In the following section, we explain how integer expressions are translated into logic terms. One obstacle to overcome is that there are several integral types in Java but only the logic type \texttt{int}. Given the following sequent

\[ \Longrightarrow (i = i + 1; i > 0) \]
and assume \( i \) is a program variable that had been declared to be of program type `long` (the logic type of \( i \) is `int`). As one of summands is of program type `long` the addition is widened to type `long`, i.e., it results only in an overflow (or underflow) if the normal mathematical addition results in a value outside of the range of type `long`.

The assignment is obviously side-effect free and can be directly moved into an update. Modeling the Java semantics faithfully, the application of the according assignment rule should result in

\[
\{ \text{i := addJLong(i, 1)} \} \text{\(i > 0\)}
\]

using the addition with overflow function for `long`. However, performing this step within KeY results instead in the sequent

\[
\{ \text{i := javaAddLong(i, 1)} \} \text{\(i > 0\)}
\]

where the Java operator `*` has been translated using the function `javaAddLong : int \times int \rightarrow int` (if \( i \) would have been declared of program type `int` the function `javaAddInt : int \times int \rightarrow int` would have been used).

These functions are intermediate representations which are used to represent the translation result of an integer program operation. The reason that we support three different integer semantics to cater for different usage scenarios. The three semantics are

- **Integer ignoring overflow semantics:** All Java integral types and their operations are interpreted as the normal arithmetic operations without overflow. In this semantics the function `javaAddLong` would be interpreted as the arithmetic addition on \( \mathbb{Z} \) (the same holds for `javaAddLong`). On the calculus level the corresponding rule would simply rewrite `javaAddLong(t1,t2)` to `add(t1,t2)`. This semantics does obviously not model the real program semantics of Java and is hence unsound and incomplete w.r.t. to real-world Java programs. It is nevertheless useful for teaching purposes to avoid the complexities which stem from modulo operations.

- **Java integer semantics:** Java integer semantics is modeled as specified by the JLS, i.e., some operations might cause an overflow. This means `javaAddLong` would be interpreted the same as `addJLong`. In this semantics the axiom rule for `javaAddLong` simply replaces the function symbol by `addJLong`. This semantics is sound and (relatively) complete, but comes with higher demands on the automation and requires in general more user interaction.

- **Integer prohibiting overflow semantics:** This semantics provides a middle ground between the two previous semantics. Intuitively, when using this semantics one has to show that all arithmetic operations are safe in the sense that they do not overflow. In case of an overflow, the result is a fixed, but unspecified value. Verifying a program with this semantics ensures that either no overflow occurs, or in case of an overflow, the value of this operation does not affect the validity of the property under verification (i.e., the property is true for any outcome of the
operation). This semantics is sound, but only complete for programs that do not rely on overflows.

The predicate symbols \textit{inByte, inChar, inShort, inInt} and \textit{inLong} which determine if the provided argument is in the value range of the named primitive type are also interpreted dependent on the chose integer semantics. While the ignoring overflow semantics always interprets these predicates as true, the two other semantics evaluate define them to be true if the given argument is within the bounds of the primitive type (for instance, \textit{inInt}(x) is true in the latter semantics iff \( x \geq -2^{31} \land x < 2^{31} \) is true).
References


References


Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006. (Cited on page 64.)


Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Ererke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated...*
References


Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung. The KeY approach for the cryptographic verification of Java programs: A case study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012. (Cited on page 594.)


References


References


Stijn De Gouw, Jurriaan Rot, Frank S. De Boer, Richard Bubel, and Reiner Hähnle. OpenJDK’s java.util.Collections.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina Pasareanu, editors, Computer Aided Verification - 27th International Conference,


Ullrich Geilmann. Formal verification using Java’s String class. Studienarbeit, Chalmers University of Technology and Universität Karlsruhe, November 2009. (Cited on page 161.)


Christoph Gladisch. Verification-based test case generation for full feasible branch coverage. In Antonio Ceroni and Stefan Gruner, editors, *Proceedings, Sixth IEEE International Conference on..."


Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamaric. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In Mauro Pezzè and Mark Harman, editors,


References


Ran Ji. Sound program transformation based on symbolic execution and deduction. PhD thesis, Darmstadt University of Technology, Department of Computer Science, 2014. (Cited on pages x, 482, 491 and 492.)


Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006. (Cited on pages 219 and 293.)


José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, Automated Reasoning, Second
References


André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning,
References


Steffen Schlager. Handling of integer arithmetic in the verification of Java programs. Diplomarbeit, University of Karlsruhe, July 10 2002. (Cited on pages 230 and 245.)


References


References


